Efficient Parallel Computing for Solving Linear Systems of Equations

Kevin P. Allen*

September 23, 2003

Abstract Linear systems of equations are common throughout the disciplines of science. The conjugate gradient method is a common iterative method used to solve systems with symmetric positive definite system matrices. With a matrix-free implementation, the method is optimal with respect to both memory usage and performance, and we are able to solve problems that are much too large for single processor computers. Using a high-performance Myrinet interconnect, excellent speedup is possible for at least up to 32 processors. This illustrates the power of parallel computing in solving large problems much faster than on a single processor.

1 Introduction

This paper reports on computational experiments on the brand-new IBM cluster in the Department of Mathematics and Statistics here at UMBC. It is an IBM 1350 cluster with 64 processors arranged in 32 dual-processor nodes with 2.0 GHz Intel Xeon chips (512 KB L2 cache) and 1.0 GB of memory per node. These nodes are connected with a high-performance Myrinet interconnect. The 32 nodes appear to be small computers stored on a rack each with 2 processors, connected together by a 32 port Myrinet switch. Additional information on this computer is available at `http://www.math.umbc.edu/~gobbert/kali.html`.

A parallel computer allows for multiple central processing units (CPUs) to work on a computation simultaneously. In essence, a parallel computer is having a group of people (in our case, up to 64 people) split a problem up and each work on a part while still communicating results to each other.

Parallel computing allows data to be distributed over a number of nodes in order for each processor to work on a smaller portion of the overall data. The network of computers share data and are able to communicate with each other sharing results. This sharing allows us to decrease the time taken to solve problems, as well as allowing the solution of much

*Department of Mathematics and Statistics, University of Maryland, Baltimore County, 1000 Hilltop Circle, Baltimore, MD 21250 (kallen1@math.umbc.edu).

larger problems. This combined power of each processor should help us in making the program faster than one that just runs on 1 processor. The two main components of parallel computing are:

We want to solve problems faster. Speedup is defined as the ratio of the time taken to complete the job using 1 process and the time taken to complete the job using p processes [7]. For optimal speedup using p processes, we should obtain a factor of speedup of p. For example if the code was run on 1 and 2 processes and we computed the speedup, optimal speedup would occur when the speedup was equal to 2. This would show that the 2 processes case ran in half the time that it took the 1 process case to finish. If we ran on p processes, optimal speedup would be achieved when we reached a speedup factor of p between the 1 process run and the p processes run. This enables us to compute solutions to problems that once took hours on 1 processor to taking minutes using parallel computing.

We want to solve larger problems. Memory is a limiting factor in the size of the problems that we are able to solve. Larger problems take up more memory. There comes a point where we would like to solve problems that are much larger than 1 processor can handle. Parallel computing allows us to split the data into portions that do not consume most of the memory allotted for each processor. Therefore, we have extra memory that could be used to make our problem larger.

This study tests a brand-new computer by code with known behavior to show that it performs well. Research goals still exist though, since we want excellent performance on this caliber of a machine.

2 Numerical Method

The focus of the computation is to solve a classical prototype problem, the Poisson equation with a homogeneous Dirichlet boundary condition, using finite differences in two dimensions. This yields a linear system with a symmetric positive definite system matrix. This property of the system matrix allows the use of the conjugate gradient method as the linear solver.

The conjugate gradient method is implemented in the programming language C using the Message Passing Interface (MPI) standard [6] for communications between parallel processors. It coincides with various textbook definitions for the algorithm [8]. This iterative method for the solution of a linear system of equations of dimension n involves 1 matrix-vector product, 2 dot products, and 3 vector updates (saxpy operations) per iteration. Our implementation is optimized with respect to memory usage by storing exactly 4 vectors of length n, the smallest number possible, and by using a function that implements the matrix-vector multiplication in matrix-free form, i.e., does not store any matrix. The vectors stored are the approximation to the solution x, the search direction p, the residual $r := b - Ax$. and the auxiliary vector q. The method is programmed in generic form, as the problem only enters via the right-hand side as input and the system matrix contained in the matrix-vector multiplication function.

Each vector $U = (U_k)$ of length $n = N^2$ is split across the p processors by cutting its representation $u_{ij} = U_k$ in the y-dimension. This results in N/p rows of x-data to be stored on each processor. The parallel form of one conjugate gradient algorithm requires 4 communication operations per iteration. To compute the matrix-vector product, processors need to interchange one x-row of data with their neighboring processors below and above. These communications are implemented with nonblocking `MPI_Isend`/`MPI_Irecv` commands. Since the dot products apply to vectors split across the processors and since the results are needed on all processors, a `MPI_Allreduce` operation is required in each of the 2 dot products. The serial part of the dot products, local to each processor, is implemented in naive C code.

The most expensive operation in each iteration of the conjugate gradient or other Krylov subspace methods is the matrix-vector product $V = AU$ of the system matrix $A \in \mathbb{R}^n$ with a vector $U \in \mathbb{R}^n$.

Storing all n^2 elements of $A \in \mathbb{R}^{n \times n}$ explicitly is known as "dense storage mode." Since most of the n^2 elements are 0, this is wasteful both in memory, as a lot of zeros are stored, and in runtime, as many multiplications involving zeros are needlessly computed. Simulations using this storage mode are available in the software package Matlab and can be easily programmed in a source-code language like C.

A common idea is to take advantage of the sparsity of the matrix, i.e., the low percentage of non-zero elements. In this "sparse storage mode," only non-zero elements are stored along with integer index information to indicate the position of the element in the matrix. This reduces the memory requirements for our system matrix to approximately $5n$ in 2-D, not counting any integer arrays, which are significant, e.g., in Matlab. This also improves performance, as only multiplications with those elements are computed that are stored explicitly, i.e., that are non-zero. This implementation is readily available in Matlab or in the parallel toolbox library PETSc [4].

To reduce the memory usage further, we take advantage of the constant coefficients in pre-determined positions of the system matrix. A function is provided that accepts a vector $U \in \mathbb{R}^n$ as input and returns the vector $V = AU \in \mathbb{R}^n$ as output; each component V_k is computed as summation of the appropriate components of U multiplied with hard-coded coefficients. This matrix-vector multiplication function is inserted at the appropriate place of our implementation of the conjugate gradient method in the programming language C. This technique is known as a "matrix-free implementation," because no elements of A are stored at all. It is the most efficient approach to memory usage.

3 Results

Memory is an important limitation to the size of the system we can solve. Aside from the system matrix, the vectors of length n used in the method make up the bulk of the memory. We ignore any other shorter vectors or variables. The estimation is simply made by counting the number of vectors used. As stated earlier, the minimum number of vectors needed to compute the solution of a linear system using the conjugate gradient method is 4

of length n. A prediction of $4n$ is almost all of the memory we need to compute the solution with the matrix-free method. Using the sparse storage mode, pieces of the system matrix need to be stored. Since our matrix is pentadiagonal, we can assume that only 5 vectors of length n are stored for the system matrix. The memory prediction is just $5n$ added to the matrix-free implementation prediction. Densely storing the system matrix requires a vector of n^2 elements added onto the matrix-free implementation prediction. Table 1 shows the theoretical predictions for how much memory would be used by the C code using the dense, sparse, and the matrix-free implementation, respectively when double-precision arithmetic is used. The sparse prediction is included for comparison with an implementation of this problem using Matlab [5]. For convenience, the column n displays the number of degrees of freedom $n = N^2$ for each value of N.

For our C code, the predicted values for the dense and matrix-free implementations in Table 1 compare well to observed values. Implementations in PETSc and Matlab using sparse storage show a large difference between predicted and observed values: Overhead in using PETSc could be the cause for the much larger observed values of memory usage of 42, 145, and 565 MB for $N = 512$, 1024, and 2048, respectively. Observing that Matlab's `pcg` function sets up several additional real and integer vectors of length n, both of which are stored in double precision, explains its larger memory usage.

Less than 1 MB of memory is used for a sparse or matrix-free computation compared to 134 MB when we are storing the system matrix for $N = 64$. Doubling the value to $N = 128$, we see that it takes over 2 GB using dense storage. This is not possible to compute this on our machines because we only have 1 GB of memory available for 1 processor computations, so $N = 64$ is the largest system we can accommodate. Clearly, a matrix-free implementation is the best setup.

Now that we have demonstrated that we can solve larger problems, we can show that we can solve them faster. For the studies up to 32 processors, only 1 CPU per node was used. This way the node would not have to share resources for each CPU, in hopes that this would benefit performance.

Table 2 and Figure 1 summarize the timings and speedups for this cluster, using up to 64 processors. It is readily apparent from both Table 2 and Figure 1 that the timings continue to improve significantly all the way up to 32 processors. Looking at our largest case, $N = 4096$, it takes just over 4 minutes to run on 32 processes as opposed to a little over 2 hours on 1 process. We can solve a system of linear equations with over 16.7 million degrees of freedom in roughly 4 minutes. Any better-than-optimal speedup is due to an over efficient 1 processor case.

Using 64 processors, a significant slow-down is observed. This is the first time that both CPUs on a node are used. Figure 1 shows the far from optimal performance at 64 processes. The cache for each processor is a good size for non-computational uses, but does not hold much data. This leads to frequent loading of data from memory to the cache. The nodes have a 32-bit bus which can not serve the data as fast as the processors can use it.

Systems of higher values of N can be solved in parallel on this cluster. The parameters used in this study though limit their results. For $N = 8192$, the method has to take around 12,000 iterations to compute a convergent solution. Time also becomes an issue when solving

these much larger problems. For the largest possible system, $N = 32768$, we would have over 1 billion degrees of freedom. Estimating the possible runtime of this system shows that it may take over 30 hours to compute a solution. With preliminary setup still being conducted, it is not feasible to consume so much of the resources available by this cluster at this time.

To analyze the importance of the interconnect hardware for this algorithm, we will look at results from another cluster that has only a fast ethernet interconnect. The same computations are performed on a cluster with 4 dual-processor nodes with 1.0 GHz Intel Pentium III processors (256 KB L2 cache) and 1.0 GB of memory. This cluster is connected using a 100 Mbps ethernet interconnect. Though the processor speed is less than that of the cluster used earlier, our results are scalable.

Table 3 and Figure 2 show the results for up to 8 processes. We see that the use of about 4 processors constitutes the most efficient use of the resources. This result shows that a high-performance network is required for this algorithm.

4 Conclusions

Parallel computing is a useful tool for solving large problems faster. The reduction in memory due to the matrix-free implementation not only allows for the solution of much larger systems, but faster run times. The parallel performance studies show that speedup is obtained for the two-dimensional problem. To obtain excellent speedup for at least up to 32 processors, a high-performance interconnect like a Myrinet connection is necessary. These observations reflect the fact that the conjugate gradient method involves 4 communications per iteration and therefore requires a tight coupling of the cluster. The results of this performance study show how important it is to have 64 processors available. Up to 32 processors, the speedup is very good, but using 64 processors, we can see the end of perfect speedup. Thus we able to test the code beyond its limit of effectiveness.

Preliminary studies with this code on an identical machine can be found in [2] and [3]. A more indepth explanation of the method and further results can be found in [1].

Acknowledgments

I would like to thank the Provost's Office for their undergraduate research award for the academic year 2003–04. The hardware used in the computational studies was partially supported by a SCREMS grant from the National Science Foundation with additional support from UMBC. See `http://www.math.umbc.edu/~gobbert/kali` for additional information.

References

[1] K. P. Allen, *A parallel matrix-free implementation of the conjugate gradient method for the Poisson equation.* Senior thesis, University of Maryland, Baltimore County, 2003.

[2] K. P. Allen and M. K. Gobbert, *A matrix-free conjugate gradient method for cluster computing*. Submitted.

[3] ——, *Coarse-grained parallel matrix-free solution of a three-dimensional elliptic prototype problem*, in Computational Science and Its Applications—ICCSA 2003, Part II, V. Kumar, M. L. Gavrilova, C. J. K. Tan, and P. L'Ecuyer, eds., vol. 2668 of Lecture Notes in Computer Science, Springer-Verlag, 2003, pp. 290–299.

[4] S. Balay, W. D. Gropp, L. C. McInnes, and B. F. Smith, *PETSc users manual*, Tech. Rep. ANL–95/11 — Revision 2.1.3, Argonne National Laboratory, 2002.

[5] *MATLAB Release 12.1 (Version 6.1)*. The MathWorks, Inc., Natick, MA.

[6] Message Passing Interface Forum, *MPI: A message-passing interface standard*, International Journal of Supercomputer Applications, 8 (3–4) (1994).

[7] P. S. Pacheco, *Parallel Programming with MPI*, Morgan Kaufmann, 1997.

[8] D. S. Watkins, *Fundamentals of Matrix Computations*, Wiley, second ed., 2002.

N	n	dense	sparse	matrix-free
16	256	< 1	< 1	< 1
32	1,024	8	< 1	< 1
64	4,096	134	< 1	< 1
128	16,384	2,184	1	< 1
256	65,536	34,362	5	2
512	262,144	549,760	19	8
1,024	1,048,576	8,796,100	75	34
2,048	4,194,304	140,740,000	302	134
4,096	16,777,216	2,251,800,000	1,208	537

Table 1: Predicted memory usage of the two-dimensional problem for the three storage methods in Megabytes (MB).

N	1	2	4	8	16	32	64
512	16	8	4	2	< 1	< 1	< 1
1,024	107	54	33	17	8	4	3
2,048	868	436	226	116	69	36	38
4,096	7,249	3,744	1,794	959	476	245	289

Table 2: Timings in seconds for the two-dimensional problem for 1, 2, 4, 8, 16, 32, and 64 processes on the 64-processor cluster with a Myrinet interconnect.

N	1	2	4	8
256	8	4	2	2
512	59	32	18	14
1,024	466	259	133	104
2,048	3,879	1,960	1,024	784
4,096	31,301	15,783	8,170	6,282

Table 3: Timings in seconds for the two-dimensional problem using nonblocking sends and receives for 1, 2, 4, and 8 processes on the 8-processor cluster.

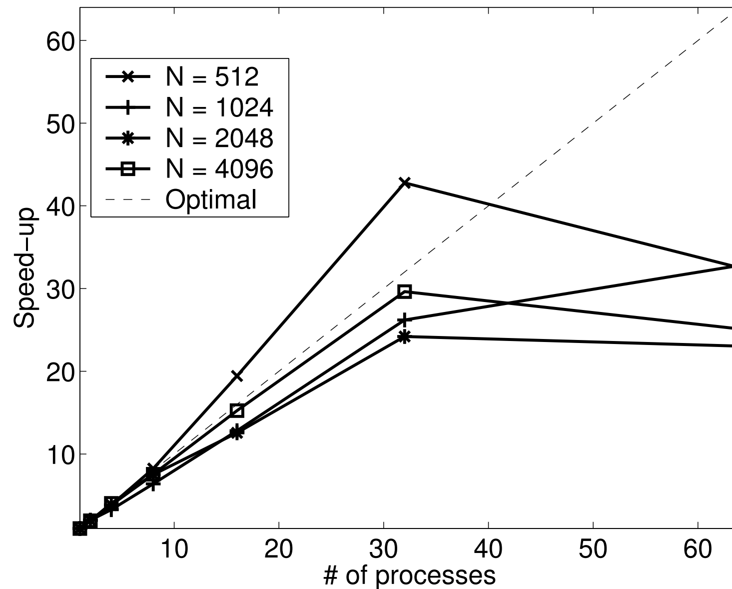

Figure 1: Plot of the observed speedup of the conjugate gradient method on the two-dimensional problem for $N = 512$, 1024, 2048, and 4096 using no BLAS and nonblocking communication on the 64-processor cluster with a Myrinet interconnect. The optimal case of linear speedup is shown for comparison.

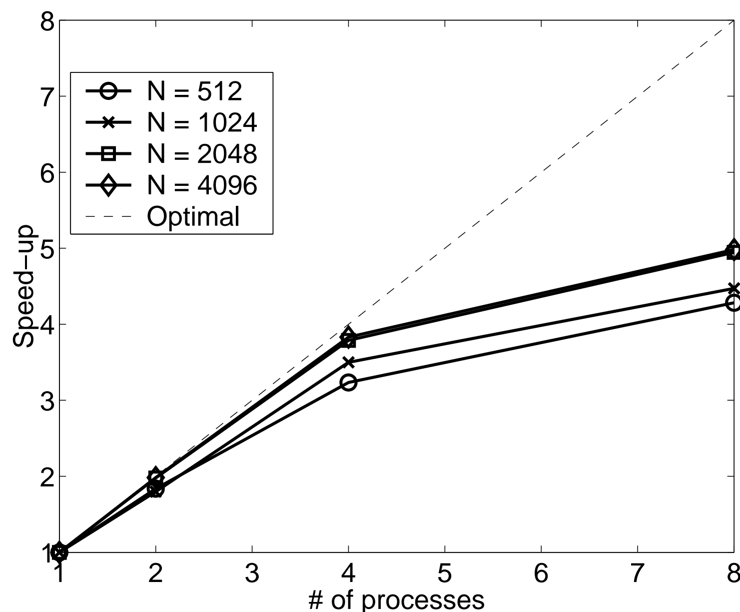

Figure 2: Plot of the observed speedup of the conjugate gradient method on the two-dimensional problem for $N = 512$, 1024, 2048, and 4096 using nonblocking communication on the 8-processor cluster. The optimal case of linear speedup is shown for comparison.